BRAC UNIVERSITY

Inspiring Excellence

Department of Computer Science and Engineering

Course Code: CSE341	Credits: 1.5
Course Name: Microprocessors	Semester: Spring'25

Lab 03

Flow control instructions and Branching Structures

I. Topic Overview:

For assembly language-programs to carry out useful tasks, there must be a way to make decisions. In this lab, students will familiarize themselves with how decisions can be made with the jump instruction. The jump instructions transfer control to another part of the program. This transfer can be unconditional or can depend on a particular combination of status flag settings. After introducing the jump instructions, we'll use them to implement high-level language decision structures.

II. Lesson Fit:

There is a prerequisite to this lab. Students must have a basic idea on the following concepts:

a. Flag register
b. Some basic operations such as MOV, ADD, SUB, MUL and DIV
c. Basic I/O operations
d. Character encoding using ASCII

III. Learning Outcome:

After this lecture, the students will be able to:

a. Control flow of the program
b. Write conditional statements in assembly

IV. Anticipated Challenges and Possible Solutions

a. Students may find it difficult to visualize how the program control flow changes when using jump

Solutions:

i. Step by step simulation

b. Directly coding in assembly may come off as challenging.

Solutions:

i. Writing the pseudocode first then then converting it to assembly may help

V. Acceptance and Evaluation

Students will show their progress as they complete each problem. They will be marked according to their class performance. There may be students who might not be able to finish all the tasks, they will submit them later and give a viva to get their performance mark. A deduction of 30% marks is applicable for late submission. The marks distribution is as follows:

Code: 50%

Viva: 50%

VI. Activity Detail

a. **Hour: 1**

Discussion: Jump instruction

In programming, execution normally follows a **sequential order**—one instruction after another. However, sometimes we need to **change the sequence** of execution based on **conditions or other requirements**. To achieve this, we use jump instructions. Jump instructions allow us to:

- ➔ Implement **decision-making**
- ➔ Create **loops**
- ➔ **Skip unnecessary** code

Types of Jump Instructions

Type	Description
Unconditional Jump (JMP)	Directly jumps to a specific part of the code, skipping intermediate instructions.
Conditional Jump (e.g., JZ, JNZ, JE, JNE, etc.)	Jumps based on a condition (e.g., if a variable is zero, if two values are equal, etc.).

Unconditional jump

When writing programs, sometimes we need to move from one part of the code to another without executing everything in between. We can tell the computer to skip some instructions and continue execution from a different section. An **unconditional jump** moves the execution directly to a labeled section of the code **without any condition**. This is useful when certain instructions **do not need to run every time**.

Syntax: JMP code_label

Here, **code_label** is the point in the code where execution will continue.

Example:

```
MOV AX, 5
MOV BX, 8

JMP my_line   ; Skip the next two instructions
MOV AH, 4
MOV DL, 6

my_line:
MOV DL, 7   ; Execution continues from here
```

How it works: The program starts by setting values in AX and BX. The instruction JMP my_line moves execution directly to my_line, skipping MOV AH, 4 and MOV DL, 6. Execution resumes at my_line, where MOV DL, 7 is executed. Here my_line is the code label to jump to.

Conditional Jumps:

A **conditional jump** allows a program to make **decisions** based on specific conditions. Unlike an **unconditional jump (JMP)**, which **always** moves execution to a label, a **conditional jump** only occurs **if a certain condition is met**.

Conditional jumps are used in:

- ➔ **Decision-making (if-else)**
- ➔ **Loops (for, while, do-while)**
- ➔ **Handling different program flows efficiently**

```
if (x>5):

// code
```

In the line if (x>5), a comparison is done between the content of x and 5. The decision whether to execute the enclosed code or not depends on the result of the comparison.

In assembly the comparison is done by a piece of code called **CMP**.

Syntax: CMP destination, source.

- The comparison is done by **destination - source**.
- The result is not saved anywhere but affects the flags.
- The result of subtraction can be 0, AX > BX or AX< BX.
- Example:

```
MOV AX, first_number
MOV BX, second_number
CMP AX, BX; AX- BX
```

The comparison is done in the third line. Now there could be 5 possibilities AX==BX,

AX> BX, AX< BX, AX>= BX, AX<= BX. The decision is based on one of these options,

and is performed by "jump" instruction, denoted as "J".

Condition	Jump Instruction	Explanation
AX == BX	JE	jump if destination and source are equal
AX > BX	JG	jump if destination > source
AX < BX	JL	jump if destination < source
AX >= BX	JGE	jump if destination >= source
AX <= BX	JLE	jump if destination <= source

But jump where???

Jump in that line which we want to execute when one of the above conditions is satisfied. How will we get the line number? We do not have to worry about the line numbers because we will name the line(s) ourselves.

`x = 5;`
`if (x > 4) :`
`  //code`
	`MOV AX, 5 ; Load 5 into AX`
	`CMP AX, 4 ; Compare AX with 4 (5 - 4)`
	`JG My_Line ;Jump to "My_Line" if AX>4`
	
	`; (If AX <= 4, execution continues here)`
	
	`My_Line:`
	`; Code to execute if AX > 4`

Note the declaration of the line name ends with a colon (:).

Please go through page number 96 of the book to know about more conditional jumps.

Branching Structures

In high-level languages, branching structures enable a program to take different paths, depending on conditions. There are **three common branching structures.** In this section, we'll look at three structures.

IF-THEN :

- A condition is evaluated (True or False).
- If True, execute the statements inside THEN.
- If False, skip the statements and continue execution.

The IF-THEN structure may be expressed in pseudo code as follows:

```
If CONDITION is TRUE
  Then
        excute true branch STATEMENTS
End-If
```

The condition is an expression that is true or false. If it is true, the true-branch statements are executed. If it is false, nothing is done, and the program goes on to whatever follows.

Example: Replace the number in AX by its absolute value

Solution: A pseudocode algorithm is:

```
IF AX < 0
THEN
   Replace AX with -AX
END_IF
```

It can be coded as follows:

```
;if AX < 0:
    CMP AX, 0;     AX < 0?
    JGE END_IF;    If no, then exit
;then
    NEG AX;        If yes, then change the sign
END_IF:
```

How it works: The condition AX < 0 is expressed by CMP AX,0. If AX is greater than or equal to 0, there is nothing to do, so we use a JGE (jump if greater than or equal) to jump around the NEG AX. If condition AX < 0 is true, the program goes on to execute NEG AX.

Problem Task: Task 01 (Page 8)

b. **Hour:**

Discussion: Branching Structures (Cont.)

IF-THEN-ELSE:

- Evaluates a **condition**.
- If **True**, executes the **IF block**.
- If **False**, executes the **ELSE block**.

The IF-THEN structure may be expressed in pseudo code as follows:

```
If CONDITION is TRUE
  Then
       excute true branch STATEMENTS

Else
       execute false branch STATEMENTS
```

The jmp instruction comes in use when if - else condition is employed.

Let us see the use of conditional and unconditional jumps together in a program where we will find the greater of 2 numbers.

JAVA

```
System.out.println("enter the first number");
int x = k.nextInt();
System.out.println("enter the second number");
int y = k.nextInt();
if (x>y){
      System.out.println(x+" is greater");
}
else{
      System.out.println(y+" is greater");
}
```

PYTHON

```
print("enter the first number")
x = int(input())
print("enter the second number")
y = int(input())
if (x > y):
   print(x + " is greater")
else:
   print(y + " is greater")
```

Assembly

```
data segment
  ; add your data here!
  pkey db "press any key...$"
  a db "Enter first number$"
  b db "Enter second number$"
  c db " is larger$"
ends
stack segment
  dw 128 dup(0)
ends
code segment
start:
  ; set segment registers:
  mov ax, data
  mov ds, ax
  mov es, ax
  ; add your code here
  lea dx, a ;Print a
  mov ah,9
  int 21h

  mov ah,1
  int 21h
  mov bl,al ;move input to bl

  lea dx, b ;Print b
  mov ah,9
  int 21h

  mov ah,1
  int 21h
  mov cl,al ;move input to cl

  cmp  cl,bl  ;compare  the  two
  inputs
  jg line1

  mov dl,bl
  mov ah,2
  int 21h
  lea dx, c
  mov ah,9
  int 21h
  jmp e

line1:
  mov dl,cl
  mov ah,2
  int 21h

  lea dx, c
  mov ah,9
  int 21h

e:
  lea dx, pkey
  mov ah, 9
  int  21h  ;output  string  at
  ds:dx

  ;wait for any key....
  mov ah, 1
  int 21h

  mov  ax,  4c00h  ;exit  to
  operating
  system.
  int 21h
ends
end start ; set entry point and
stop the assembler.
```

Discuss why the "JMP" statement is crucial in this program.

- Prevents Unnecessary Execution
- Ensures Correct Program Flow

CASE:

A **CASE** is a multiway branch structure that tests a register, variable, or expression for particular values or a range of values. It allows a program to execute different blocks of code based on the value of an expression. Instead of multiple **IF-THEN-ELSE** conditions, the **CASE structure** provides a **clean and efficient** way to handle multiway branching. The general form is as follows:

```
CASE expression
   Values_1: statements_1
   Values_2: statements_2
   ...
   Values_n: statements_n
END_CASE
```

- ➔ **The `expression` is tested** and matched against predefined values
- ➔ **If a match is found**, the corresponding block executes. If its value is a member of the set values_1, then statements_1 are executed
- ➔ **Each value set is unique** (no overlapping values).
- ➔ **If no match is found**, execution continues to the next instruction after `END_CASE`.

Problem Task: Task 02 – 09 (Page 8)

Comparison: IF-ELSE vs. CASE

IF-ELSE	CASE
Repeats conditions multiple times	Evaluates the expression once
Slower for multiple conditions	Faster for multiple conditions
Harder to read and maintain	More structured and readable
Example: Multiple `if (x == value)`	Example: `CASE x`

c. **Hour: 3**

Discussion:

Check progress while the students carry on with the rest of the tasks.

Problem Task: Task 10-14 (Page 9)

VII. Home Tasks: All the unfinished lab tasks

Lab 4 Activity List

Task 01

Take a number in AX, and if it‟s a negative number, replace it by 5.

Task 02

Suppose AL and BL contain extended ASCII characters. Display the one that comes first in the character sequence.

Task 03

If AX contains a negative number, put -1 in BX; if AX contains 0, put 0 In BX; if AX contains a positive number, put 1 in BX.

Task 04

If AL contains 1 or 3, display “o”; if AL contains 2 or 4 display "e".

Task 05

Read a character, and if it's an uppercase letter, display it.

Task 06

Read a character. If it's "y" or "Y", display it; otherwise, terminate the program.

Task 07

Write an assembly program to check whether a number is even or odd.

Task 08

Write a program to input any alphabet and check whether it is vowel or consonant.

Task 09

Write a program to check whether a number is divisible by 5 and 11 or not.

Task 10

Write a program to find the maximum and minimum between three numbers.

Sample execution:

User input : 2 3 4

Output: Maximum number is 4

Minimum number is 1

Task 11

Write a program that takes as input all sides of a triangle and check whether triangle is valid or not. If the sides form a triangle, print “Y”, otherwise print “N”.

Task 12

Write a program that takes a digit as an input and outputs the following. If the digit is within 0-3, it prints “i”, If it‟s within 4-6, it prints “k”, If it‟s within 7-9, it prints “l” and if it‟s 10, it prints “m”.

Task 13

Write a case to print the name of the day of the week. Consider the first day of the week is Saturday.

Sample execution:

User Input: 3

Output: Monday

Task 14

Write a case to print the total number of days in a month.

Sample execution:

User Input : 3

Output: 31 day